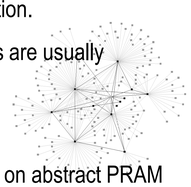
Accelerating CUDA Graph Algorithms at Maximum Warp

Pervasive Parallelism Laboratory, Stanford University
Sungpack Hong, Tayo Oguntebi, Kunle Olukotun

Graph Algorithms

Introduction: Graph Algorithms

- Many interesting problems are constructed using graphs: social networks, supply chain analysis, genealogy, …
- However, those problems are often immense in size: even simple algorithms take significant amounts of time on such large graphs
- Parallelism is required for faster execution.
- Challenge: Interesting graph instances are usually irregularly shaped .



Conventional Approaches

- Theory researchers have concentrated on abstract PRAM (Parallel Random Access Machine) algorithms; however, real commodity machine implementations are rare.
- Clusters are not favored due to their huge communication overhead. Partitioning irregular graphs is also difficult.
- PRAM machines have been implemented as supercomputers (e.g. cray-xmt). But they are expensive and hard to access.

Previous Work: Graph Algorithms on GPU

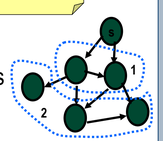
- Implementations of PRAM algorithms on GPUs [1, 2]
- Key observation: GPU architecture is a miniature replica of a PRAM supercomputer.
  - ➔Let each thread process one parallel task. (e.g. one thread per node operation)
  - ➔Exploit parallelism and higher memory bandwidth.
- Problem: Performs badly for irregular graphs
- Example: Find BFS ordering of nodes from a source node.

```
__global__ BFS_Kernel(…) { n = tid; …
 if (order[n] == curr_level)
    for( j = nbrs[begin[n]] to nbrs[end[n]])
       if (order[j] == INF)
          order[j] = curr_level + 1
}
```

(Frontier expansion method)

➔Algorithm consists of multi-staged kernel calls

➔Each stage expands frontiers by one level



Two major issues for irregular graphs

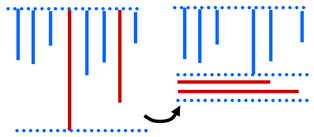
[A] Execution path divergence (varying numbers of neighbors)

[B] Scattered memory access pattern (no address coalescing)

Our approach: Methods for efficiently addressing irregularly shaped graphs

Deferring High Fan-out

- Detect large-sized works and put those works into a queue
- Deferred works are processed in subsequent kernel calls.



- Concurrent queues can be implemented reasonably cheaply, if they either only grow or decrease during a given phase.

[pros] Less imbalance
[cons] Multiple kernel calls

```
_device_ Add_Queue (Q, n, items)
{  old = atomicAdd(Q.idx, n)
   cpy( &Q.data[old], items, n)
}
```

Dynamic Workload Distribution

- Implemented in conjunction with warp-based execution
- Instantiate only as many Thread-Blocks as the number of SMs
- Dynamically allocate workloads per warp from a work queue

[pros] Prevents SMs from stalling for one long-running warp
[cons] Work queue overhead

Warp-Based Execution

- Each warp processes a chunk of nodes, serially
- A warps utilize its threads for SIMD operations. (Or for very short SIMT operations only)

[pros] Coalesced memory access.
No stall due to divergence
[cons] Under-utilization **unless enough SIMD operations**

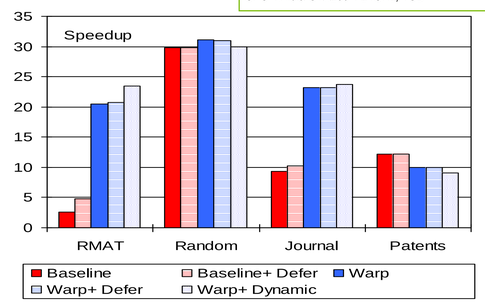
```
_global_ BFS_kernel2(…) { _shared_ sOrder[];
  n = w_id = tid / (WARP_SZ);
  w_off = tid % (WARP_SZ); .
  SIMD_CPY(sOrder, order[n], WARP_SZ, w_off);
  for( i = 0 ; i < WARP_SZ; i++)
    if (sOrder[i] == curr_level) {
      SIMT_update_nbr(nbrs, begin[n], end[n],
                      w_off, …)
} }
```

```
_device_ SIMT_update_nbr (...)
{ for (i= begin+w_off; i<end; i+=WARP_SZ) {
    j = nbrs[i];
    if (order[j] == INF)
       order[j] = level + 1
} }
```

Results

Performance Comparisons

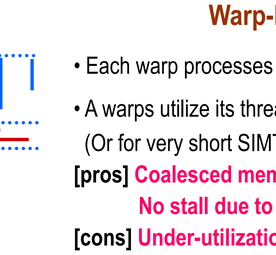
CPU: Intel Xeon E5345 2.33GHz, 32GB
GPU: Nvidia GTX260 1.24GHz, 1GB



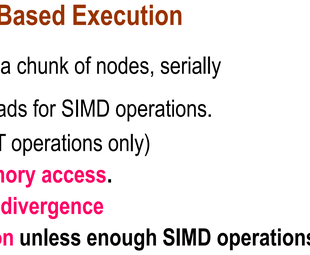
- Speedup over single-threaded CPU version
- Our baseline GPU implementation is optimized over [1] by 20%

Properties of Input Graphs

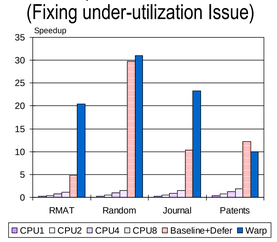
	Type	# Nodes	# Edges	Shape
RMAT	Generated [1]	4 x10 ^{6}	48 x10 ^{6}	Irregular
Random	Generated [1]	4 x10 ^{6}	48 x10 ^{6}	Regular
Journal	Real-world data [3]	4,308,451	68,993,773	Irregular
Patents	Real-world data [3]	1,765,311	10,564,104	Regular



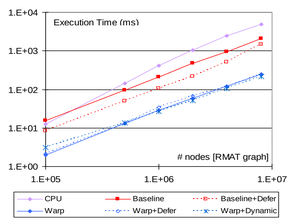
Sub-warp based execution (Fixing under-utilization Issue)



Effect of defer-threshold



Comparisons to multi-threaded CPU implementation [4]



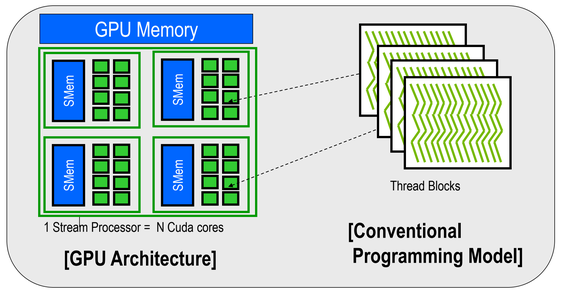
Execution time over size of graph

Summary

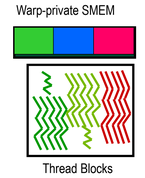
[A] Warp-based execution results in good speed-up for irregular graphs

[B] Underutilization issue can be addressed by using Sub-warps

Rationale: Warp-Based Execution



- **Actual thread execution is not exactly PRAM**
  Threads in a warp ➔ unit of SIMD
  Warps in a thread-block ➔ unit of SMT
  Thread-blocks ➔ unit of multi-processing
- **Therefore…explicitly utilize those traits**
  - Assign a chunk of jobs to each warp.
  - Each warp performs those jobs serially.
  - Each warp uses its 32 threads to perform SIMD operations.
  - Each warp can own a private SMEM partition.
  - Synchronization is inherent.



[Warp-based Programming Model]

Future Work

- **Further study of effectiveness of warp-based approach**
  - On other graph algorithms
  - On other applications having similar workload imbalance issue
- **Automatic Generation for Warp-Based Kernels**
  - User description ➔ SIMD kernels and calls (e.g. SIMD update_nbr in our example)
  - C Macros or Compiler front-end

References

[1] P Harish and P Narayanan, Accelerating Large Graph Algorithms on the GPU using CUDA [HiPC 2007]

[2] F Dehne and K Yogaratnan, Exploring the Limits of GPUs with Parallel Graph Algorithms [arxiv preprint 2010]

[3] Stanford Network Analysis Platform, http://snap.stanford.edu

[4] Small-world Network Analysis and Partitioning, http://snap-graph.sourceforge.net/

Contact information

E-mail

- {hongsup, tayo} @ stanford.edu